# 论文1: AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration (23年6月挂arXiv)

## 作用：

权重重要性是有区别的，要去识别保护重要的权重。

## 研究背景：

LLM是通才模型，使用校准集的量化补偿方案可能会导致模型过拟合，从而失去泛化性（如GPTQ），通过保留异常值为FP16的方法不会减少计算时间。

## 研究问题：

* 主要比较对象是GPTQ，认为GPTQ可能会在校准集上过拟合，提出了一种activation-aware的weight量化。
* 提出之前分离异常值的方案会增加计算时间，设置比原始FP16的计算时间还长，需要一种全新的权重保护方案。

## 研究层次：

* 软件级
* PTQ，Weight-only量化。但量化是基于激活值大小的。

## 研究方向：

LLM+PTQ量化 **强相关**

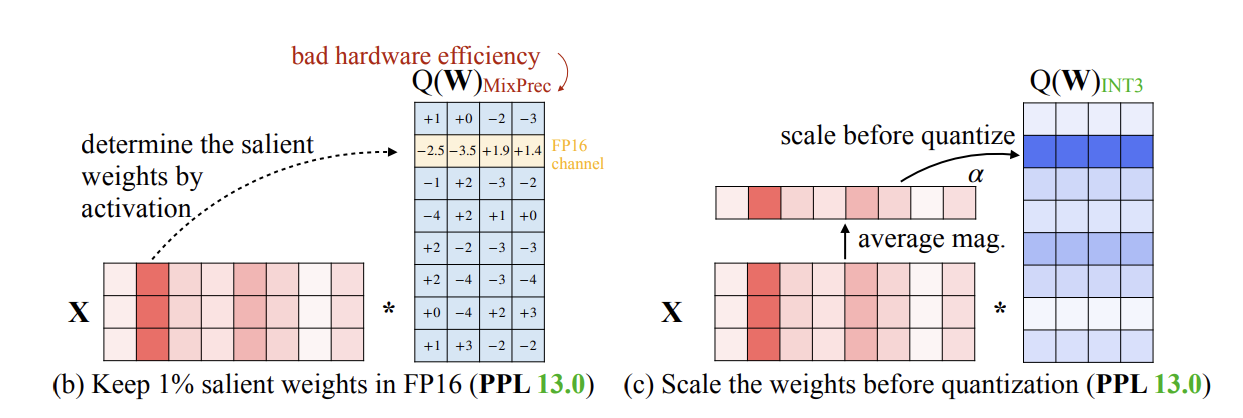
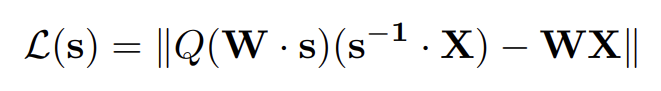
## 研究差异：

* 同：和smoothquant（相同一作）,LLM-INT8一脉相承的文章，都是识别并保护重要权重。另外AWQ使用校准集，但不做量化补偿。
* 异：重要权重的识别方法以及保护重要权重的手段不同。

## 挑战：

* QAT不实用，GPTQ使用校准集去做量化补偿会降低模型的泛化性能（GPTQ每量化一列需要对其他所有列做量化补偿）。因为 LLM 是通才模型。
* 从weight角度识别异常值从而保留的做法有局限性，因为weight本来就很好量化，应该从激活值的角度考虑。
* 识别到重要权重后如何保护是个问题，直接保留成原精度无法提高计算时间，减少推理延迟。

## 文章提出的解决方案:

* AWQ不做量化补偿也能有很好的性能，这是因为AWQ使用了Activation-aware去量化权重
* activation-aware是指观察激活值矩阵最大的前1%的数所对应的weight行（之前的文章是直接找weight矩阵前1%大的异常值）进行权重保护。
* 
* 对于保护权重的方法，作者认为以FP16保存不能够加速计算，且硬件不友好，所以它的方式是寻找一个缩放比例s，在参数量化之前乘以这个比例，计算时输入X除以这个比例，以减小误差 

就是求这个最优化这个loss的s值。但是量化方法又不可导，所以得想一个其他的方法来计算出这个s。 SmoothQuant有一个观点，认为W的大小幅度实际上远小于Activation的大小幅度， 假如把两者进行平衡，用一个缩放系数s乘W的进行放大，再把X除以s进行缩小，之后再做量化的消耗和量化损失会更小。回到求s这个问题上来，作者把s分成两个值S\_x和S\_w相乘，我们根据以上的公式，W越大的时候需要的s越小，X越大，需要s越大，来达到平衡两者幅度的效果，于是有：

fig: 至于里面的和，是两个权重系数，取多少是作者网格搜索搜出来的，最后发现beta其实很小，说明重要性很低，主要是activation在起作用（所以取名叫AWQ)

**总结**： AWQ其实是延续了smoothquant的思路，转移量化难度，但此处是把weight量化难度转移到激活，因为激活保持不量化，所以对激活进行scale操作不太会影响最后精度。

# 论文2: SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression (2023年6月挂arXiv)

## 作用：

和AWQ不同的重要权重识别和保护策略，可以和AWQ相互参考。

## 研究背景：

LLM量化难度相较于以往的模型更加困难，因为其中存在重要权重。参数的对模型的重要程度，存在极强的不均衡性。1%的参数，可能主导的量化过程中损失的性能，假如我们在量化中保护这1%的参数，就能极大程度保护模型的性能不受影响。

## 研究问题:

LLM存在重要参数，这些参数非常重要不能直接删去或截断，如何识别并保护这些重要参数是需要解决的问题。

## 研究层次:

软件级，PTQ **强相关**

Weight-activation量化

## 研究方向:

LLM+PTQ量化

## 研究差异：

* 同：和AWQ一样，基于参数重要性不均衡去识别并保护重要参数。
* 异：实现方式上有差异，如何识别和如何保护重要参数都有区别。

## 挑战:

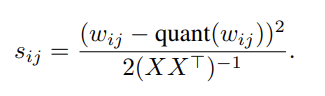
SpQR提出的视角在于如何使手机等边缘设备能够运行大模型，因此主要挑战是如何压缩模型使其能够适应此类设备，并且保持足够精度。

## 文章提出的解决方案:

1.权重的识别：

对于每一层，它使用一个小的输入数据集X，用来计算单个参数被量化前后造成的的误差. 有了之后再取top 1%的参数认为它们是重要参数进行保护。





2.权重保护：

在挑选出参数之后，SqQR使用一个稀疏矩阵来单独保存这些参数，令这些重要参数的精度仍为fp16。

除此之外，SqQR在实验中还观察到重要参数往往以行或者列聚集（异常值的通道聚集性），因此提出使用更小的group\_size比如8或16，而非GPTQ中常用的128.

# 论文3: OMNIQUANT: OMNIDIRECTIONALLY CALIBRATED QUANTIZATION FOR LARGE LANGUAGE MODELS (23年8月挂arXiv)

## 作用：

GPTQ + AWQ = OmniQuant，在搜参方法上可以借鉴。

## 1 研究背景：

LLM需要数十亿参数大量内存，存储和计算需求过高，且参数中存在重要异常值。量化难度较以前高。GPTQ和AWQ方法的结合者。

## 2 研究问题:

本文旨在解决大型语言模型的存储和计算需求过高的问题，作者本人说研究的问题是能否用PTQ达到QAT的表现，并因此提出了一种新的量化技术，OmniQuant。

## 3 研究层次:

* 软件级,PTQ
* Weight and Activation 量化

## 4 研究方向:

LLM+PTQ量化+搜参 **强相关**

## 5 研究差异：

* 同：GPTQ的block量化过程+AWQ的量化难度迁移
* 异：量化过程和难度转换过程中的参数不再人为指定，而是抽取出来变成可学习的变量。

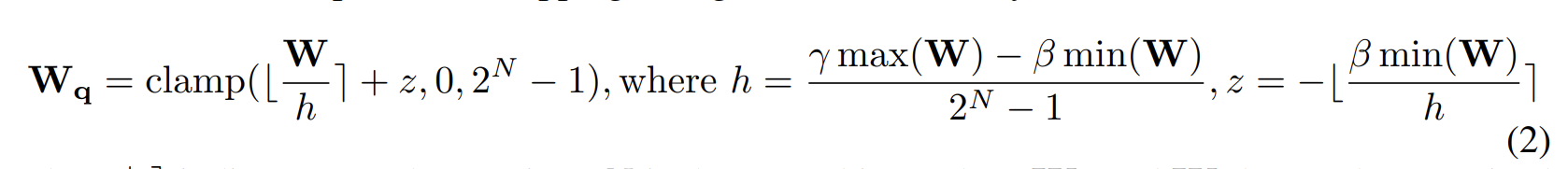
## 6 挑战:

1. 由于离群通道的存在，激活很难量化。考虑到weight分布是扁平和均匀的，SmoothQuant和 Outlier Suppression+ 通过将量化困难从激活迁移到weight，并预先设定迁移强度来解决这一问题。
2. 由于与激活相对应的权重的重要性，权重的量化误差在最终性能中也起着关键作用。 SqQR和 OWQ 以全精度保留关键权重，而 AWQ使用网格搜索通道缩放来保护这些权重。
3. 上述工作在迁移强度和缩放因子等量化参数上的选取相当粗放，基本上是手工指定+写死，导致低于4bit的量化效果不理想（2/3 bit）。

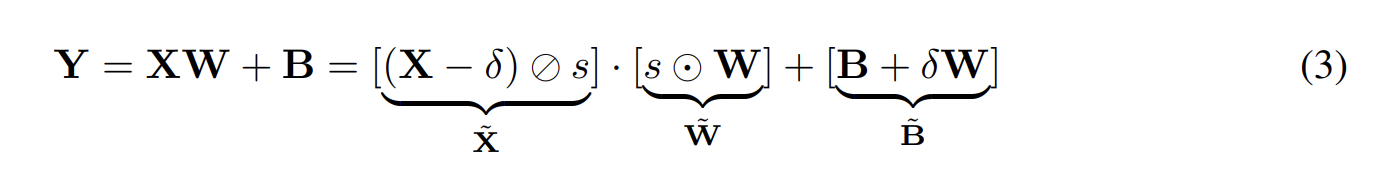
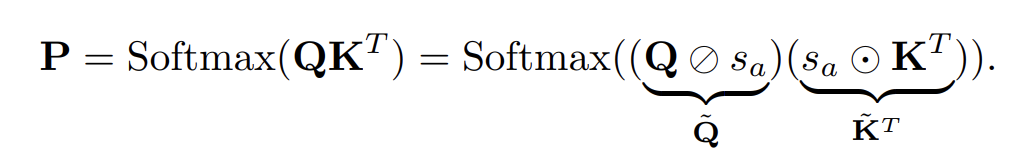
## 7 文章提出的解决方案:

**LWC 可学习量化裁剪：**

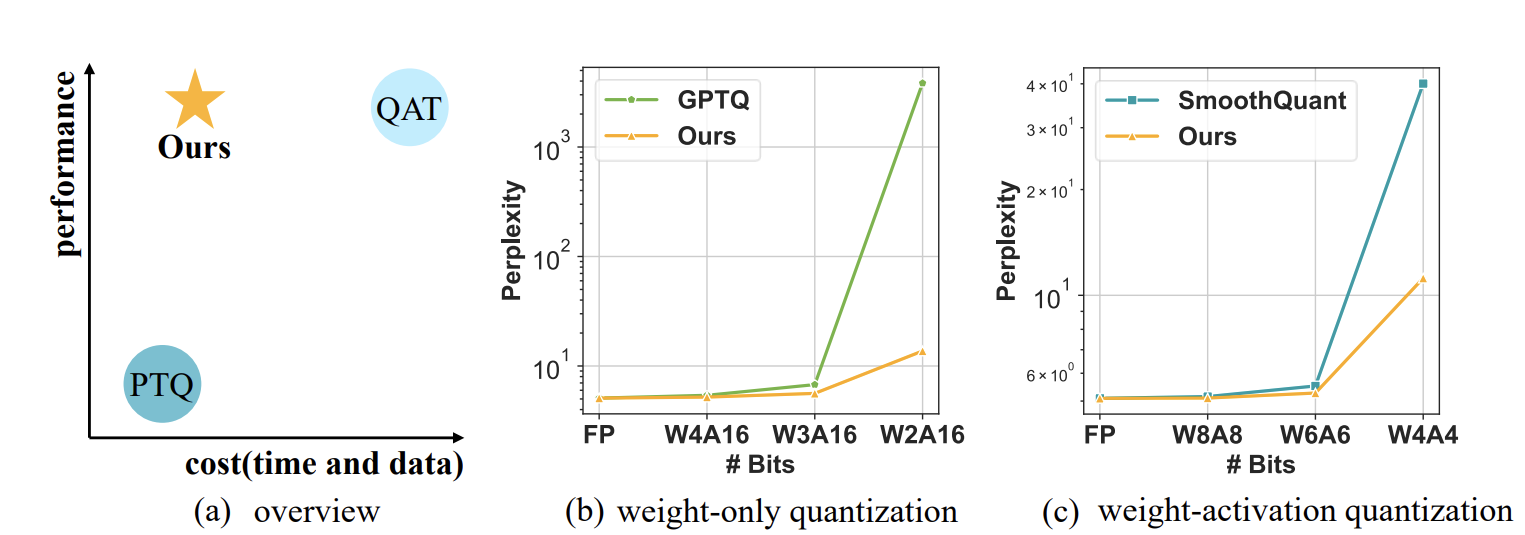
* 之前GPTQ量化是取block里的最大值和最小值作为范围，再分成2^n个点，把每个值round到这个这些点上。omniquant提出不如做个缩放，把比方说以90%的最小值到最大值为范围。超过的就当离群点，直接round到边界点上。
* 那90%这个参数怎么定呢？做成可学习的变量。在公式里是γ和β



**LET 可学习等价转换：**

* 在SmoothQuant和AWQ里通过把Activation除一个值s，Weight乘一个值s，来让Activation的变化幅度趋于平稳，使其更易于量化的操作，OmniQuant也差不多，只是把里面的参数变成可学习的变量。
  1. linear中的实现：
  + 
  1. attention中的实现：
  + 

优势主要体现在2/3bit的低精度量化：



# 论文4: A SIMPLE AND EFFECTIVE PRUNING APPROACH FOR LARGE LANGUAGE MODELS (23年6月挂arXiv)

## 作用：

剪枝的方法很简单粗暴，可能能改进，或借鉴到量化中。

## 研究背景：

之前的剪枝sota：Magnitude Pruning剪枝方法对传统模型很有效，但在LLM上效果不好。因为Magnitude Pruning 只适用于输入信号的强度差距不大的时候（也就是几乎没有异常值）。而LLM不是这样。作者提出一个针对LLM的剪枝方法。

## 研究问题:

如何对LLM进行剪枝，使剪枝后模型的精度损失最小。

## 研究层次:

软件级，LLM模型剪枝。

基于weight和activation的剪枝。

## 研究方向:

LLM+剪枝 **弱相关**

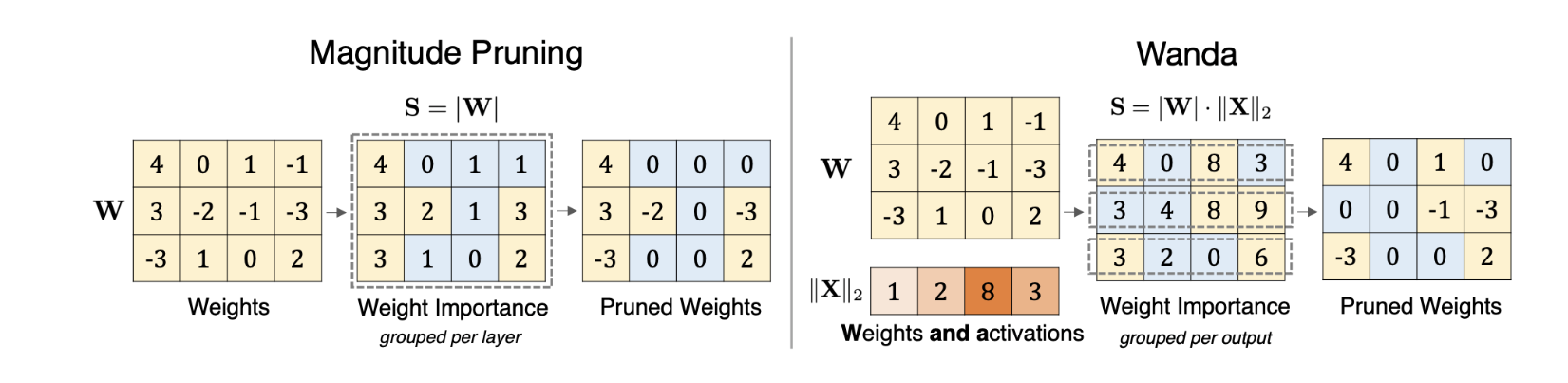
## 研究差异：

* 同：与之前的剪枝方法流程上没有区别，区别主要在剪枝粒度和识别要剪枝权重的方法。
* 异：
  1. 之前的剪枝粒度是per-layer的，对于大模型来说，这种剪枝方式可能造成通道间的剪枝不平衡（有些通道没剪，有些通道全剪了）。wanda改成per-channel。
  2. 按照激活值大小来剪。

## 挑战:

* 以前的剪枝方案不适合对异常值敏感的LLM，另外之前的方案可能需要重新训练进行权重更新以保持精度。
* per-layer粒度的剪枝方案可能造成通道间的剪枝不平衡（有些通道没剪，有些通道全剪了）。这对LLM的精度影响很大。

## 文章提出的解决方案:

* 对weight和activation乘积的矩阵，把每通道绝对值最小的50%数设为0。
* 
* 这种方案的好处在于：
* 1.考虑了激活值的大小，适合LLM。
* 2.基于通道的剪枝，相较于全局剪枝，保证了通道间剪枝的平衡性。对精度影响更小。
* 3.因为方法非常简单，只需要一次前向传递就能剪完整个模型。几乎没有额外计算。剪枝速度很快。也不需要重新训练反向传递。

相对于方案的简单性，效果还算不错，但目前在模型效率（无论是推理速度还是内存消耗）上，wanda都暂时无法带来实际的改善。这是因为稀疏矩阵版的wanda的有显存优化效果但是效果差，非稀疏矩阵版只是把值设成零该占多少显存还是多少显存。并且最终的模型困惑度，也不如目前主流的量化方法。主要优势是想法很简单，看起来有改进空间。并且和量化是正交的，可以借鉴到量化中。

**一个启发**：如果不能用稀疏矩阵计算带来实际收益，剪枝是没有必要的。

# 论文5: Efficient Memory Management for Large Language Model Serving with PagedAttention (SOSP 2023)

## 作用：

提出了pageAttention加速技术，提出了vllm框架，目前是加速llm计算最火的框架，文章深入attention的计算过程，尤其针对KV Cache做了针对性的内存优化。值得了解。

## 研究背景：

大型语言模型 (LLM) 的高吞吐量服务需要一次批处理足够多的请求，即batch要足够大才能实现高吞吐。现有系统面临着困难，因为每个请求的kv cache内存巨大，并且会动态增长和收缩。如果管理效率低下，这些内存可能会因碎片和冗余重复而被严重浪费，从而限制了能够处理的batch大小。

## 研究问题:

吞吐是 LLM 服务最重要的性能指标之一，提升 LLM 服务的吞吐可以降低服务部署成本。提升模型服务的吞吐最直观的方式是将尽量多的请求合并成一个模型输入（即 batching），使单次推理能完成尽量多的请求计算。因为对一个合并请求单次推理可以共享模型权重，多个请求推理只需访问一次权重，计算访存比提升，访存开销能被batch请求所增加的计算开销覆盖，所以单次推理合并越多请求，吞吐越高。

对于模型服务来说，单次推理最大合并请求数主要受显存制约。所以，提升模型服务吞吐这个研究话题可以转变成在显存一定的情况下，提升单次推理最大合并请求数。这也是 LLM 服务最关注的问题。

## 研究层次:

系统级，关注attention块的内存优化与加速。

## 研究方向:

LLM加速，内存优化。 **弱相关**

## 研究差异：

* 同：针对attention的加速
* 异：主要差异在如何管理attention的内存碎片问题。

## 挑战:

**Batching 策略：**

batching 能提升模型服务的吞吐。常规的模型服务的 batching 策略如下：

1. 设置一个 batching 延时以及最大 batch size；
2. 收到新请求时，将请求入队，并判断请求队列中的请求数是否达到最大 batch size 或者最早请求与最迟请求的到达时间差是否达到 batching 延时，如果条件满足，则执行第3步骤，否则继续休眠直至 batching 超时或者新请求到达；
3. 对合并后的模型输入进行模型推理，得到完整的模型结果，将结果返回客户端了，服务返回步骤2的状态，等待下一个batch的请求。

对于生成式的 LLM 模型服务来说，使用常规的模型服务 batching 策略会有两个问题：

1. **请求端到端延时提升。**论文里提到两个延时：batching 延时以及请求在队列里的等待延时。在高并发场景下，请求在队列的等待耗时约等于前序批次请求的推理耗时总和。由于大模型是自回归模型，推理耗时主要在 Decode 阶段。Decode 阶段的耗时与生成的 tokens 数（迭代次数）成正比。生成文本长度越长，Decode 耗时越长。生成单个 token 耗时在毫秒级别，那么生成千级别 token 数 Decode 耗时基本在秒级别，所以等待延时可达秒级别。线上服务基本无法接受这么长的请求等待延时。
2. **GPU计算、存储资源有效利用率低。**模型输入输出的长度不一致。文本长度不一致对模型输入和输出端的影响不同：
   * 对于模型输入而言，由于文本输入长度不同，在构造模型输入的时候需要进行 padding 预处理，那么在进行模型推理的时候就会引入对 padding 位的计算，浪费了 GPU 计算资源。
   * 推理引擎需要将 batch 内所有请求完成文本生成后，才会对下一个batch的请求进行推理。由于不同请求生成长度可能不同，在整个 batch 到达某一轮迭代时，部分请求可能已经完成推理，但此时推理引擎仍然会继续完成剩余请求推理，直至所有请求完成推理。每当一轮迭代完成一个请求的推理，意味着下一轮请求多了一个空转的请求推理，而请求队列里的请求又无法直接替代当前 batch 内已完成推理的请求使用GPU计算资源，导致GPU资源浪费。

**Attention管理策略：**

单次推理合并越多请求，吞吐越高。对于模型服务来说，单次推理最大合并请求数主要受显存制约。LLM 服务显存占用主要由三部分组成：**模型权重**、**KVCache** 以及其他**激活值**。而现有的LLM服务中，KVCache 和激活值显存开销与推理时的 batch size 成正比，batch size 越大，KVCache 和激活值显存开销越大。

作者分析了在单卡40GB 的 A100上，LLaMA-13B BF16 模型的显存占比。模型权重有 26GB，其余显存均用来存储 KVCache和激活值。设置最大batch size使 KVCache 和激活值显存占满剩余显存空间，那么 KVCache 的显存占比大于30%，激活值显存占比不及5%。

当前attention的内存分配方法是按照max\_seq\_len来固定分配一个最大内存，防止内存泄漏，但这种方式会造成内存碎片，计算效率低下，降低LLM吞吐。作者发现，**由于碎片化和过度预留，现有的系统浪费了60%-80%的内存。**

## 文章提出的解决方案:

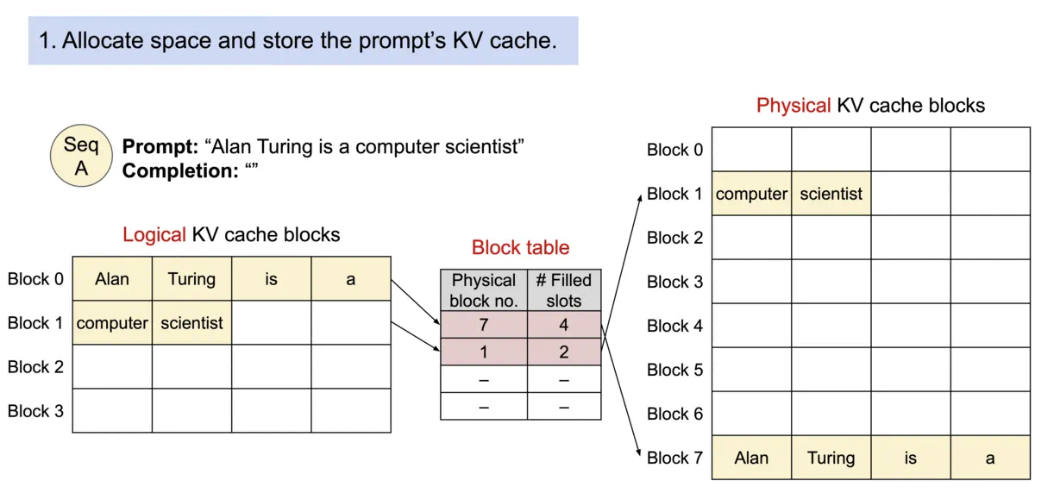
**针对常规Batching策略的不足，文章提出使用 *Iteration-Level* 请求调度方式：**

目前，传统batching策略调度被称为*Request-Level* ，与 *Request-Level* 方式相比， *Iteration-Level* 请求调度的在时间维度上粒度减小了。具体做法是，当完成一轮 Decode 迭代时，如果 Batch 内有请求完成推理（即生成 EOS token），则将该请求剔除出 Batch；如果有新的请求进入，则将新请求加入到当前 Batch 进行推理。通过这种调度优化方式，可以同时解决上述两大问题：

1. 延时方面：可以大幅度降低请求端到端延时。由于队列的请求在Decode阶段动态地插入到 Batch，所以请求的队列等待时间从原来的秒级别下降到毫秒级别（仅需等待一轮迭代即可插入到 Batch 中）。
2. 资源利用率方面：提升GPU资源有效利用率。由于队列的请求在Decode阶段每一轮结束后动态地插入到Batch，而 Batch 也在迭代过程中动态地删除已完成推理的请求，这样避免了 GPU 空转，提升资源利用率。

**针对内存占用的问题，文章提出PageAttention，使用页式内存管理来管理Attention的内存，减少内存碎片：**

PagedAttention灵感来自于操作系统中**虚拟内存**和**分页**的经典思想，它可以允许在非连续空间立存储连续的KV张量。具体来说，**PagedAttention把每个序列的KV缓存进行了分块，每个块包含固定长度的token**，而在计算attention时可以高效地找到并获取那些块。



通过块表可以自然地实现内存共享。类似进程之间共享物理页，在PagedAttention中的不同序列通过将逻辑块映射到一样的物理块上可以实现共享块。为了确保安全共享，PagedAttention跟踪物理块的引用计数，并实现了**Copy-on-Write**机制。 内存共享减少了55%内存使用量，大大降低了采样算法的内存开销，同时提升了高达2.2倍的吞吐量。